NEW VISION
INSTITUTE OF
TECHNOLOGY

Bash Script

Pre-Requisites

- **Linux Basics**
- **Command line basics**
- **Practice time**
- **No programming knowledge required**

What is Bash Script?

Bash Script is a powerful tool for automating tasks in Linux and Unix like Systems

Why Shell Scripts?

- Automate
- Time
- Productivity

Why Shell Scripts?

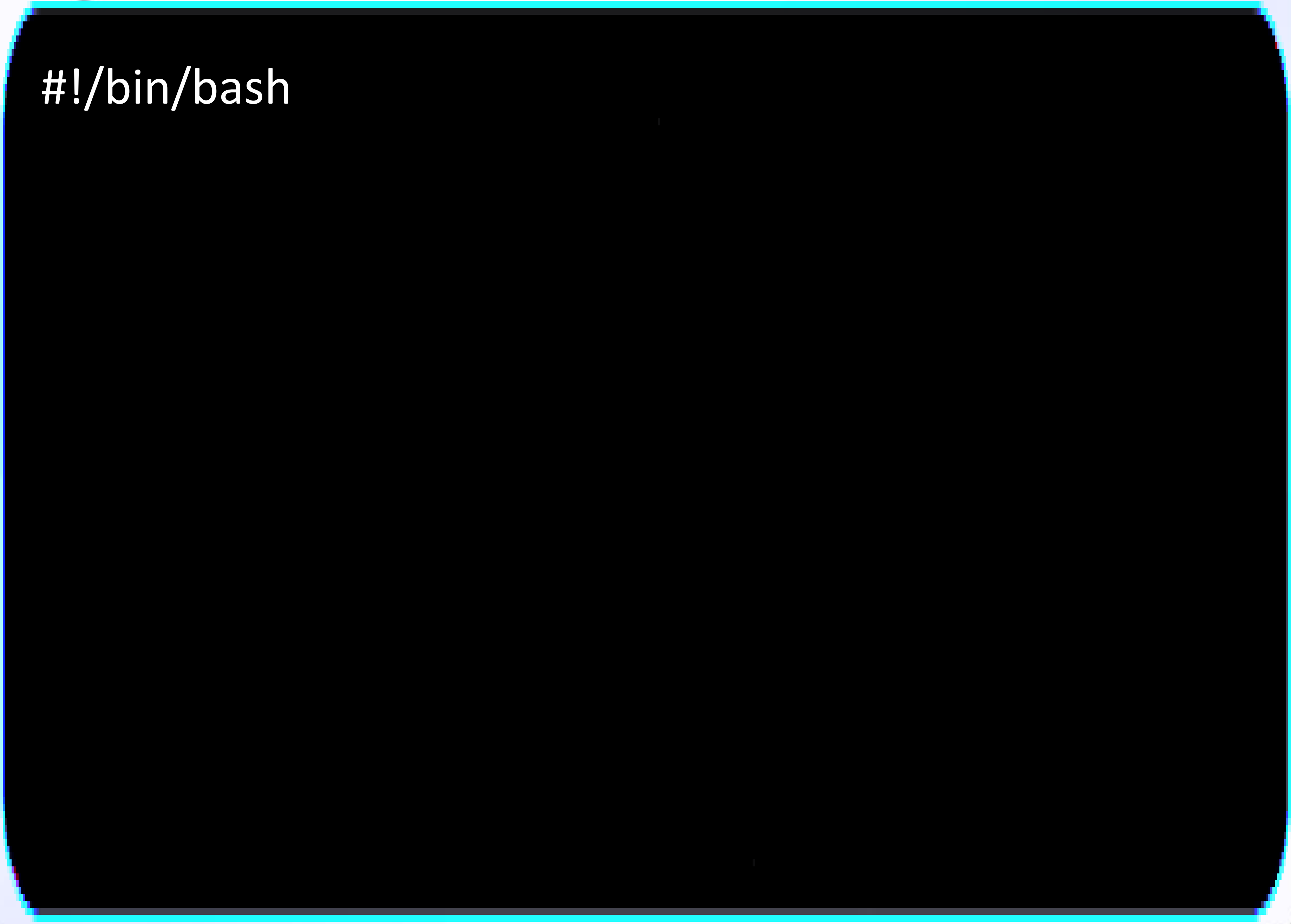
- Automate repetitive tasks like Daily Backups
- Increase efficiency and accuracy due to minimal human interaction
- Enhances collaboration and reproducibility
- Automate Installation and Patching of software on multiple servers
- Monitor system periodically
- Raise alarms and send notifications
- Troubleshooting and Audits
- Many More

Imagine a set of commands you have to run often to carry out a task. There are two ways you can do it, type and run the set of commands each time or write the set commands in a script and run the script each time you want carry out the task.

NEW VISION INSTITUTE OF TECHNOLOGY

Creating Your First Script

Bash Script



```
#!/bin/bash
```

Create file

```
$ touch first_script.sh

$ vi first_script.sh
```

In your text editor, type

```
#!/bin/bash
```

This line is the mandatory beginning of every shell script and the #! Is called Shebang. This first line tells the operating system the kind of shell is used for the scripting h sample_shell.sh

Type this and save in the text editor, then exit the editor

```
#!/bin/bash
echo “This is my first bash script!!!”
echo “This interesting and promising”
```

The shell script generally accepts commands perculiar to the shell type you are using for the script sample_shell.sh

In other words, bash shell commands are accepted in the bash script.

Bash Script

(Bash) Scripts are executable files and will therefore need the execute file permission for it to run

```
chmod u+x first_script.sh
```

or

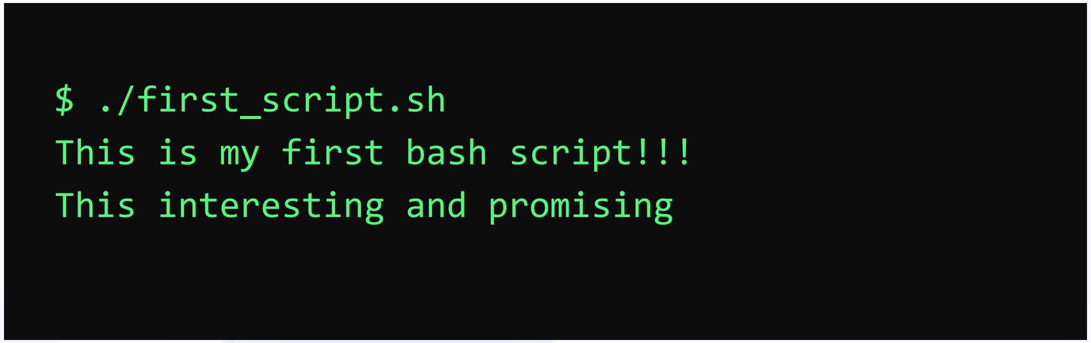
```
chmod 744 first_script.sh
```

Either of these commands gives the file the execute permission, since the default permission for new files is read, write for Owner, read for group and others

Run the script

./first_script.sh

When you run the script, it gives the output below



```
$ ./first_script.sh
This is my first bash script!!!
This interesting and promising
```

Congratulations!!!

You have successfully written and executed your first bash script

With practice, you are on your way to becoming a go-to automation Engineer

Practice Exercise

Practice Exercise

1. Write a bash script the outputs the following to standard output

I am learning bash script

Bash script is a beautiful for automation of some tasks

2. Write a bash script that list five tasks that can be automated using bash script

Naming your scripts

You should give your script descriptive names and names that makes sense. Even those who don’t know you or what your script does should see the name and suggest functions of the script

You can use any naming convention, like separating the meaningful words with underscore or using camel case etc

Bash Extension

The .sh is the extension for bash script. But your bash script can still run without an extension (which is preferred), so long as it has the shebang line and it has executable permission for the appropriate user

Best Practice

“Variable names should be in lower-case with underscores to separate words”

good:

January_2nd_backup.sh

mission_name bad:

Mission_Name

Mission Name

test.sh

Comments

Comments are very important in script.
They are used to give information about the program or script: like the authour(s) and their contact(s); the purpose/function of the script and how it works; the date and so on.

It should be possible for someone else to learn how to use your program or to use a function in your library by reading the comments (and self-help, if provided) without reading the code.

Comments are ignored during the execution of the script

Comments comment after the shebang line

Comments

To make Comments in a bash script, the # sign is used at the beginning of each of the lines of comment

```
#This is a comment
#This is another comment
```

Comments

```
############################################
# Cleanup files from the backup directory.
# Globals:
# BACKUP_DIR
# ORACLE_SID
# Arguments:
# None
############################################

function cleanup() {
 …
}

############################################
# Get configuration directory.
# Globals:
# SOMEDIR
# Arguments:
# None
```

Comments

```
# Outputs:
# Writes location to stdout
########################################

function get_dir() {
echo "${SOMEDIR}"
}

########################################
# Delete a file in a sophisticated manner.
# Arguments:
# File to delete, a path.
# Returns:
# 0 if thing was deleted, non-zero on error.
########################################

function del_thing() {
rm "$1"
}
```

Running your script

There are three ways to run your script:

1. ./name_of_script.sh

2. bash name_of_script.sh

3. The third way to run your bash script is to add the script path to the system path and then just call the script

Running your script

```
$ mkdir jetbin
$ cd jetbin
touch sample_script
```

Type and save in a text editor

```
#!/bin/bash
echo “This Script is running as if it is a build in command”
echo “because we added the script path to the system path”
```

```
$ echo $PATH
/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin
```

```
$ export PATH=$PATH:/home/Michael/jetbin
```

```
$ chmod +x sample_script
```

```
$ sample_script
```

```
$ sample_script
This Script is running as if it is a build in command
because we added the script path to the system path
```

Variables

Variables are place holders that points to assigned values.

In bash script, there are not data types. The syntax is as below:

```
variable_name=variable_value
```

There is not space before and after the equal (=) sign

Variables in bash are strings by default and interpreted as such

Variables

To call or use the defined variable,
put the dollar sign ($) before the variable name as below:

```
$variable_name
```

There is not space before and after the equal (=) sign

Variables in bash are strings by default and interpreted as such

Variables

For the shell to interpret the input as numbers, put the arithmetic operation in double parenthesis as below:

```
Sum=$((variable_name + variable_name))
```

Notice the space before and after the operator (+)

arithmetic operation

- As variable’s default type is string, to perform arithmetic operation, use the following syntax

  `$[$x+1]` or `$(($x+1))`

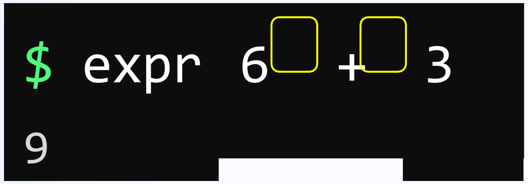
- For simpler syntax: declare variable to be numerical

  `declare –i x`

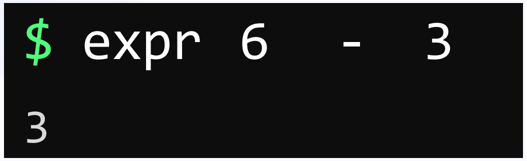
  `x=$x*10+2`

The Above are for integer arithmetic operations only

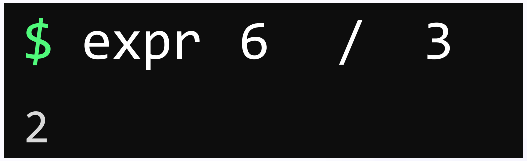
expr



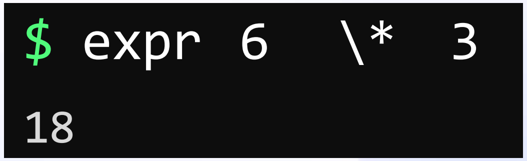
```
$ expr 6  +  3
9
```



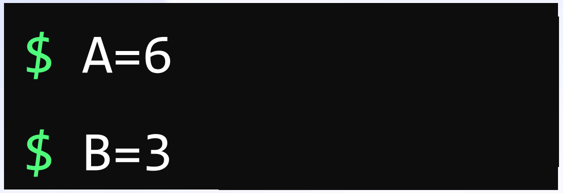
```
$ expr 6  -  3
3
```



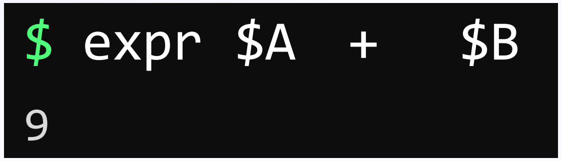
```
$ expr 6  /  3
2
```



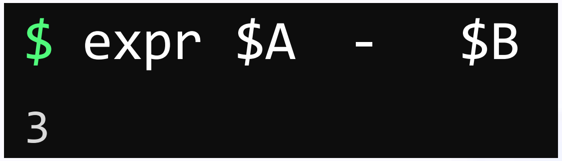
```
$ expr 6  \*  3
18
```



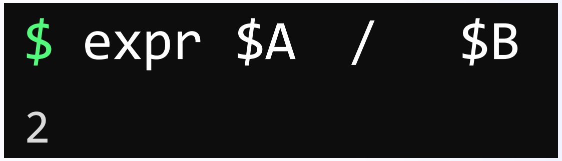
```
$ A=6
$ B=3
```



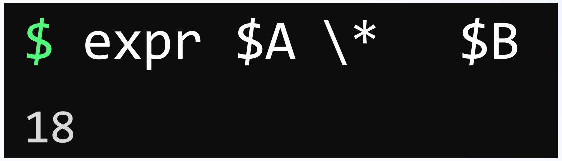
```
$ expr $A  +   $B
9
```



```
$ expr $A  -   $B
3
```



```
$ expr $A  /   $B
2
```



```
$ expr $A \*   $B
18
```

double parentheses

```
$ A=6
$ B=3
```

```
$ expr $A  +  $B
9
```

```
$ expr $A  -   $B
3
```

```
$ expr $A  /  $B
2
```

```
$ expr $A \*  $B
18
```

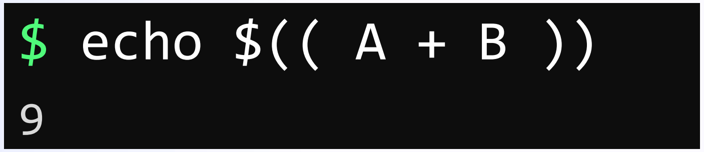
```
$ echo $(( A + B ))
9
```

```
$ echo $(( A-B ))
3
```

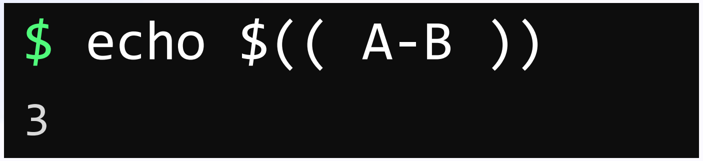
```
$ echo $((A/B))
2
```

```
$ echo $(( A * B ))
18
```

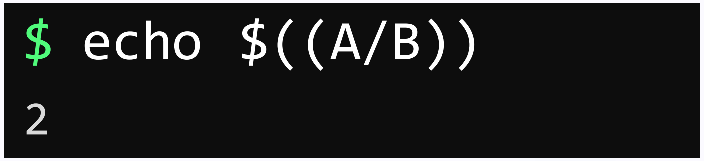
double parentheses



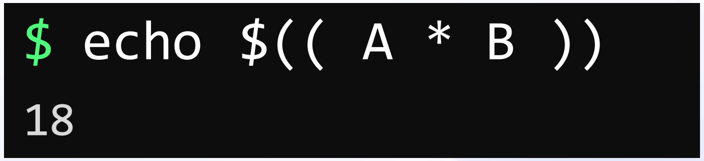
```
$ echo $(( A + B ))
9
```



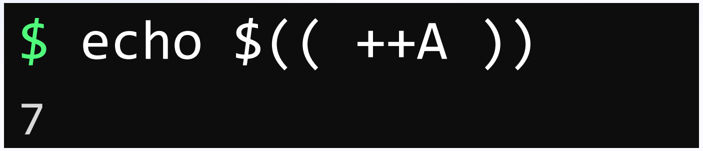
```
$ echo $(( A-B ))
3
```



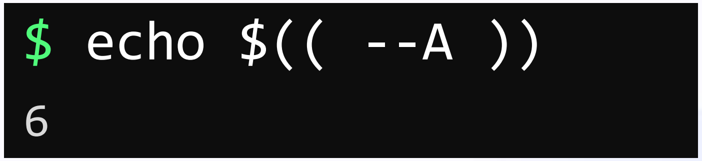
```
$ echo $((A/B))
2
```



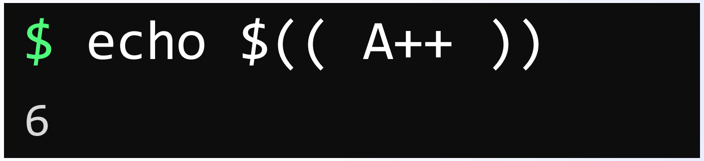
```
$ echo $(( A * B ))
18
```



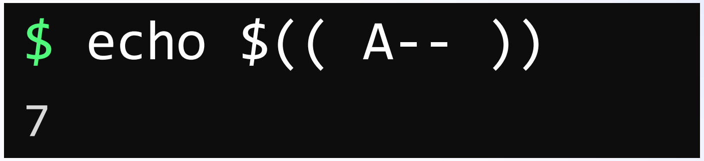
```
$ echo $(( ++A ))
7
```



```
$ echo $(( --A ))
6
```



```
$ echo $(( A++ ))
6
```



```
$ echo $(( A-- ))
7
```

Practice Exercise

Practice Exercise

Write a bash script that outputs 7 system variables with echo statements that tell what the system variable is being printed

Sample of exercise

```
#!/bin/bash

# The script is authoured by Jethro on the 21st of January, 2024
# It print 7 of the system variables

echo "Below are 7 of the System Variables"

echo “The  present user of the system is: $USER “
echo “The home directory of the present user is: $HOME
echo “The shell in use is: $SHELL
echo “The System path is $PATH”
echo “The present working directory is: $PWD
echo “The User ID of the present Account is: $UID
echo “The Host name of the system is: $HOSTNAME
```

Practice Exercise

Produce the below operations as a bash script named my_math

Run with just the name of the script (hint: you need to add the script directory to the path)

Include comments to explain what you are doing in the script

```
num1=24
num2=17

sum=$((num1 + num2))
subtract=$((num1 - num2))
multiply=$((num1 * num2))
division=$((num1 / num2))
```

Sample of exercise

```
#!/bin/bash

# The script is authoured by Jethro on the 21st of January, 2024
# It does basic mathematics and comments
# it's directory path is added to the system path and run only with the name
# Hence it displays that you can run scripts with just their names

echo "Below are the specified arithmetic operations"

num1=24
num2=17

sum=$((num1 * num2))

echo "The Sum: $num1 + $num2 = $sum"
echo "The Difference: $num1 + $num2 = $((num1 - num2))"
echo "The Multiplication: $num1 X $num2 = $((num1 * num2))"
echo "The Division: $num1 / $num2 = $((num1 / num2))"
echo "The Division: $num1 // $num2 = $((num1 / num2))"
echo "The Modulus: $num1 % $num2 = $((num1 % num2))"
```

User Inputs

Bash Script

User Inputs

Bash script takes input from the user in different ways:

1. Command Line Arguments
2. User Interactive

1. Command Line Arguments

Here, the input variables are called positional variables and are represented using the dollar sign and the number of the variable as below:

- **$1**
- **$2**

Where:

- $1 = first positional argument
- $2 = the second positional argument
- Up to $10 which is the tenth positional argument

Type and save in a text editor

```
#!/bin/bash

# This script demonstrate positional arguments

num1=$1
num2=$2

echo "for the entered arguments $num1 and $num2, the following
operations are true: "

sum=$((num1 * num2))

echo "The Sum: $num1 + $num2 = $sum"
echo "The Difference: $num1 - $num2 = $((num1 - num2))"
echo "The Multiplication: $num1 X $num2 = $((num1 * num2))"
echo "The Division: $num1 / $num2 = $((num1 / num2))"
echo "The Modulus: $num1 % $num2 = $((num1 % num2))"
```

```
./script_name 72 24
```

Type and save in a text editor

```
#!/bin/bash

# This script demonstrate positional arguments
# The scripts takes Name, age and sex of a user and adds it to a text file

echo "Name: $1" | tee -a dataform.txt
echo "Age: $2" | tee -a dataform.txt
echo "Gender: $3" | tee -a dataform.txt
echo "siccessfully saved to the dataform.txt"
```

```
./script_name MyName 27 Male
```

$* : All entered arguments
$@ : All entered arguments
$# : the number of arguments entered
$0 : script_name
$1 - $10 : corresponding names of arguments respectively

Example: saved as multivar and run with multivar 45 32 12

```
#!/bin/bash
if [[ $# = 0 ]]
then
  echo "you have not entered any variables"
else
  echo "you entered $# variables, which are: $*"
fi


for arg in $*
do
   echo argument $arg
done
```

2. User Interactive

Here, the input is gotten from the user interactively use the read command. With option p you can print an initial message to the screen as below:

```
read -p “message to display on the screen” variable_name
```

Options with read

read -p “prompt to display” variable_name
-s : (silently) used to read input without echoing to screen, used for password
-a: (array) used to read array
-t 10 : (timeout) sets a 10 seconds timeout for input, outside which the terminal exits
-n 5 : specified the number of input characters to be 5
-e : (edit) allows user to edit their input using the direction keys
-r : disables backlash escaping, hence interpreting backlashes as ordinary characters
-d “:” : (delimiter) specifies : as the delimiter to terminate input

example

```
#!/bin/bash
# This script take input interactively

read -p "enter your Name, Age and City separated by space" Name Age City

echo "Your Name is $Name"
echo "Your Age is $Age"
echo "Your City is $City"
```

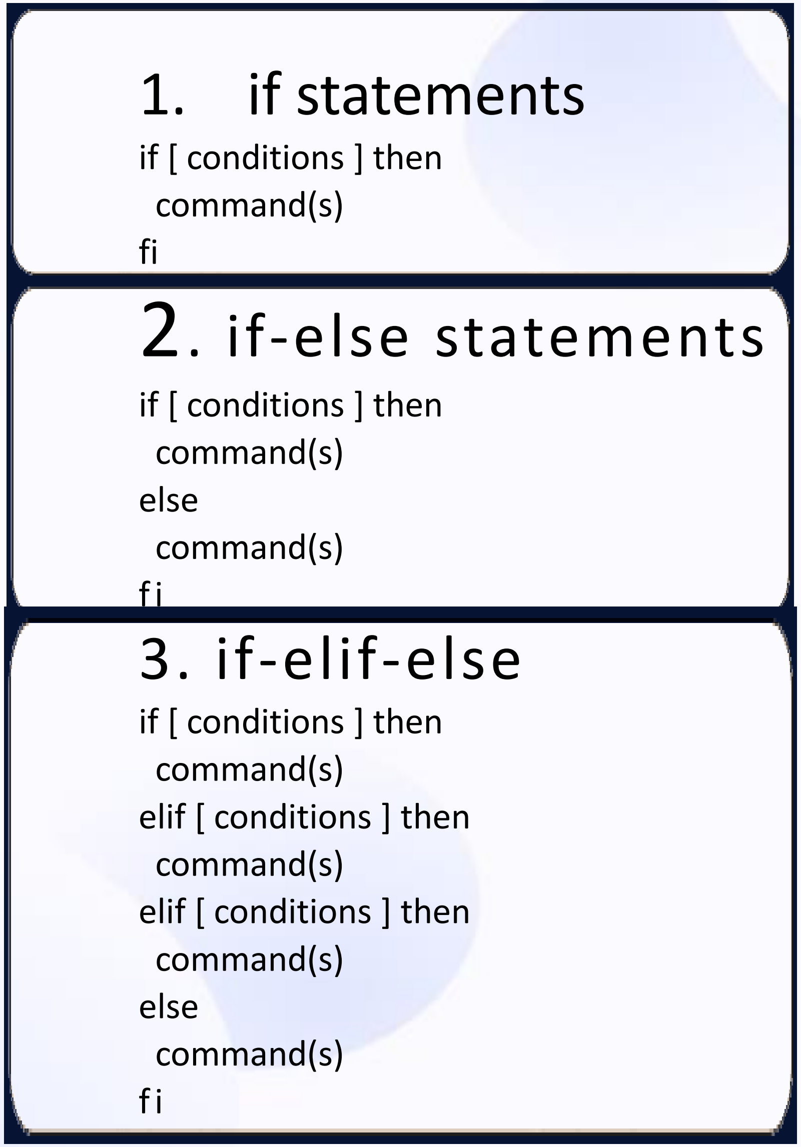
Exercise

Write a bash script that collect Name, year of birth, gender and height. Use arithmetic operator to calculate the age and then output the Name, Age, Gender and Height to the screen and to a dataform.txt

Conditional Logic

Bash Script

Conditional statements allows you to make decisions based on certain conditions in bash



1. if statements

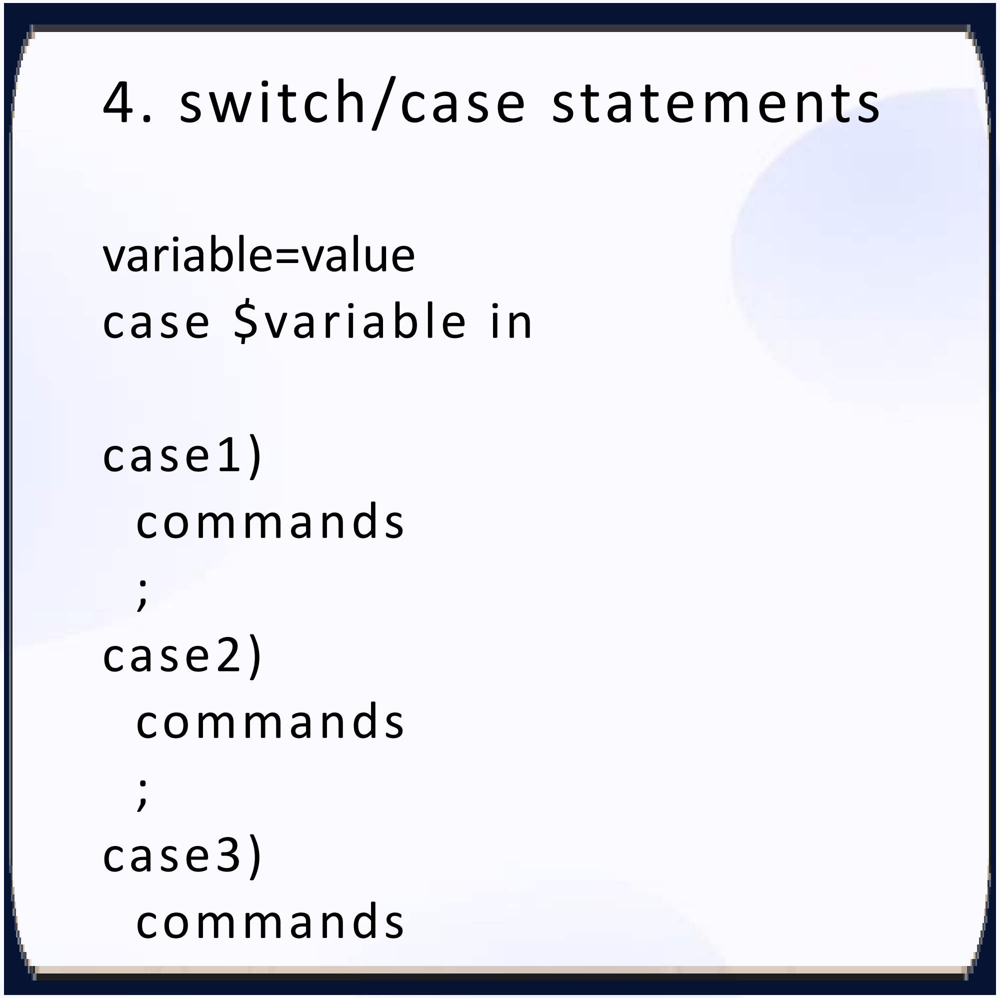
```
if [ conditions ] then
  command(s)
fi
```

2. if-else statements

```
if [ conditions ] then
  command(s)
else
  command(s)
fi
```

3. if-elif-else

```
if [ conditions ] then
  command(s)
elif [ conditions ] then
  command(s)
elif [ conditions ] then
  command(s)
else
  command(s)
fi
```



4. switch/case statements

```
variable=value
case $variable in

case1)
  commands
  ;
case2)
  commands
  ;
case3)
  commands
```

Conditional Operators

```
[ STRING1 = STRING2 ]
```

Example	Description
["abc" = "abc"]	If string1 is exactly equal to string2 (true)
["abc" != "abc"]	If string1 is not equal to string 2 (false)
[5 -eq 5]	If number1 is equal to number2 (true)
[5 -ne 5]	If number1 is not equal to number2 (false)
[6 -gt 5]	If number1 is greater than number2 (true)
[5 -lt 6]	If number1 is less than number2 (true)

Conditional Operators

```
[[ STRING1 = STRING2  ]]
```

Example	Description
`[[ "abcd" = *bc* ]]`	If abcd contains bc (true)

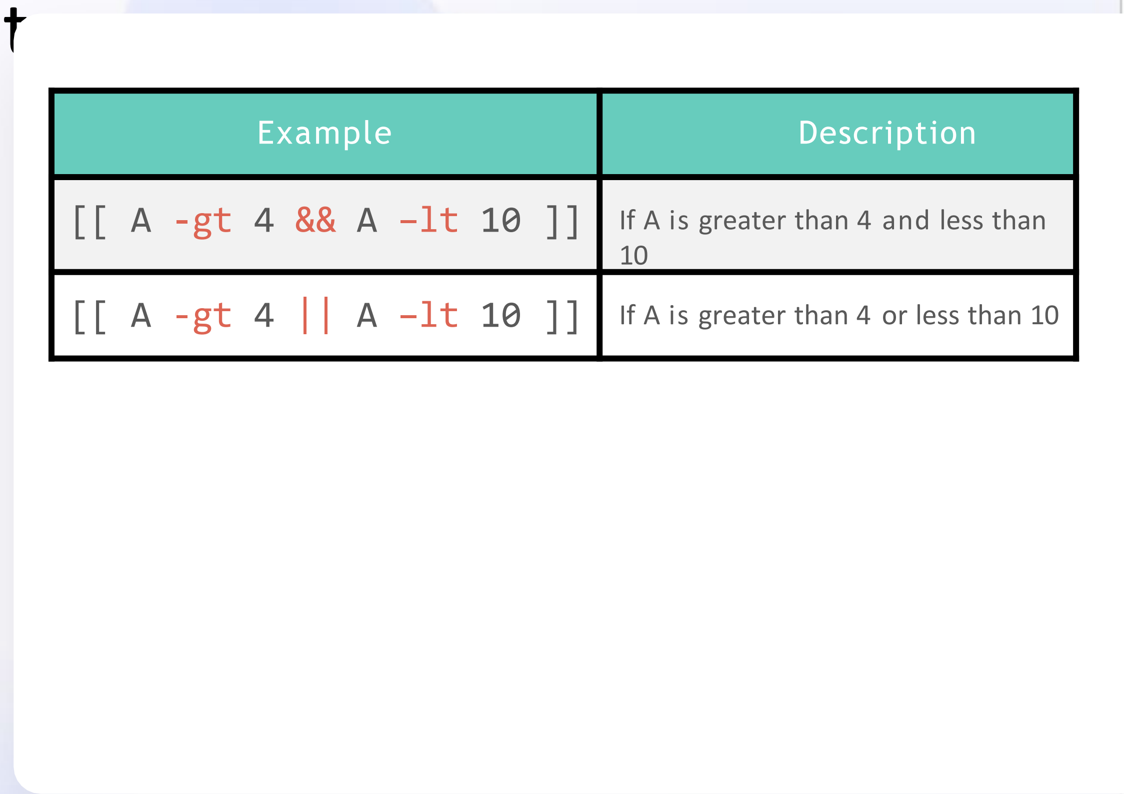
`[[ "abc" = ab[cd] ]]`
or
`[[ "abd" = ab[cd] ]]`
	If 3rd character of abc is c or d (true)

`[[ "abe" = "ab[cd]" ]]`	If 3rd character of abc is c or d (false)
`[[ "abc" > "bcd" ]]`	If "abc" comes after "bcd" when sorted in alphabetical (lexographical) order (false)
`[[ "abc" < "bcd" ]]`	If "abc" comes before "bcd" when sorted in alphabetical (lexographical) order (true)

Only in BASH

Conditional Operat

```
[  COND1    ]  &&  [    COND2    ]

[[     COND1  &&  COND2       ]]

[  COND1    ]  ||  [    COND2    ]

[[     COND1  ||  COND2       ]]
```



Example	Description
`[[ A -gt 4 && A -lt 10 ]]`	If A is greater than 4 and less than 10
`[[ A -gt 4 || A -lt 10 ]]`	If A is greater than 4 or less than 10

Conditional Operators

Example	Description
[-e FILE]	if file exists
[-d FILE]	if file exists and is a directory
[-s FILE]	If file exists and has size greater than 0
[-x FILE]	If the file is executable
[-w FILE]	If the file is writeable

Testing Strings vs Numbers

Comparing numbers

- remember (( ))
- -eq , -ne, -gt, -ge, -lt, -le

Comparing strings

- Remember [[ ]]
- Remember space after [
- =
- !=
- Unary string tests
  - [string] (not null)
  - -z (0 length)
  - -n (some length)
  - -l returns the length of the string

```
#!/bin/bash

# This script demonstrates conditional statement if
# The scripts takes Name, age and sex of a user and adds it to a text file

if [ -f dataform.txt ]; then
  #do nothing
  echo “”
else
  touch dataform.txt
fi
echo "Name: $1" | tee -a dataform.txt
echo "Age: $2" | tee -a dataform.txt
echo "Gender: $3" | tee -a dataform.txt
echo "siccessfully saved to the dataform.txt"
```

```
#!/bin/bash
# Meaning of colours
echo “Welcome to the Meaning of Colours!!!”
read -p "Enter a colour to know its meaning: " colour

if [[ $colour == "black" ]]; then
  echo "People say $colour is evil, illegality, depression, morbidity, night and death"
  echo "But it could also mean Elegance, Sophistication, Formality, Power, Strength"

elif [[ $colour == "red" ]]; then
  echo "$colour is danger, anger, chaos, please stop."
  echo "but it could also mean love, boldness, excitement, strength, determination,
courage, warmth"

elif [[ $colour == "pink" ]]; then
  echo "$colour means feminine, love, care, nurture"

elif [[ $colour == "green" ]]; then
  read -p "Enter the shade of green: " shade
  if [[ $shade == "light green" ]]; then
    echo "It is not thick agriculture."
  elif [[ $shade == "dark green" ]]; then
    echo "The agriculture is African."
  fi
```

```
#!/bin/bash

# This script demonstrates switch cases

read -p "Enter your OS: " os
case $os in
  windows)
    echo "To copy, select the text, then ctrl c";;
  linux)
    echo "To copy, select the text, then right-click";;
  macos)
    echo "To copy, you can use the windows style or the linux style";;
  android)
    echo "On Android, it is like a GUI";;
  ios)
    echo "iOS is basically MacOS on the phone";;
  *)
    echo "Unknown OS";;
esac
```

Loops

Bash Script

Loops

In bash script, loops are used to repeat command or a set of commands until certain condition(s) are met. Major types of loops:

1. for loop
2. while loop
3. until loop
4. select

for loops

The for loop repeats a set of commands for every item in a list. The syntax is as below:

```
for variable in value1 value2 … valueN
do
  command(s) to execute
done
```

```
#!/bin/bash
# This script demonstrates the for-loop

# Loop through fruits
for fruit in "Mango" "Cashew" "Apple" "Pears"
do
  echo "$fruit"
done
echo “================================================================“
# Loop through names
for name in "John" "Peter" "Jethro"
do
  echo "$name"
Done

echo “================================================================“
# Loop through numbers
numbers=(1 2 3 4 5 6 7 8)
for number in "${numbers[@]}"
do
  echo "$number"
done
echo “================================================================“
# Loop through numbers using a range
for i in {1..12}
do
  echo "$i"
done
```

mission-names.txt

```
lunar-mission
jupiter-mission
saturn-mission
satellite-mission
lunar-mission-2
mars-mission
apollo-mission
spitzer-mission
viking-mission
pheonix-mission
chandrayan-mission
gaganyan-mission
aditya-mission
nisar-mission
mangalyaan-mission
columbia-mission
challenger-mission
atlantis-mission
endeavour-mission
mercury-mission
gemini-mission
```

launch-rockets.sh

```
for mission in $(cat mission-names.txt)
do
  echo  $mission
done
```

```
for mission in $(cat mission-names.txt)
do
  echo $mission

done
```

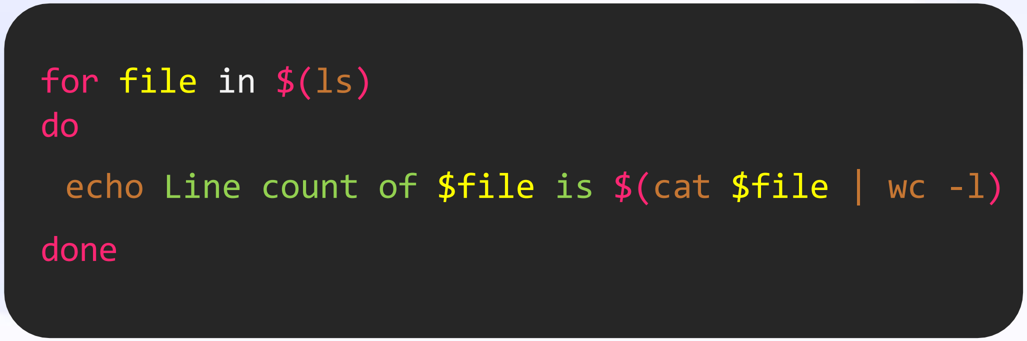
```
for mission in 1 2 3 4 5 6
do
 echo mission-$mission

done
```

```
mission-1
mission-2
mission-3
mission-4
mission-5
mission-6
```

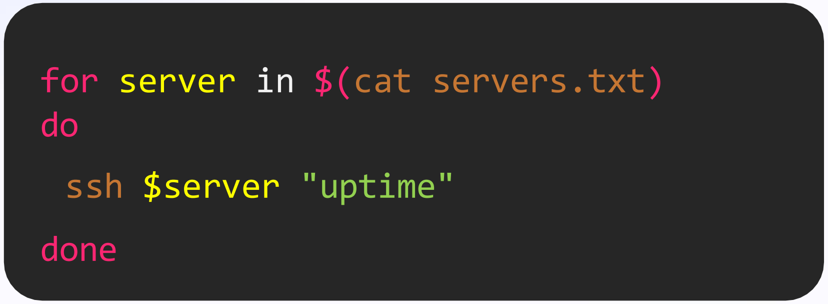
```
for mission in  {0..100}
do
 echo mission-$mission

done
```

```
mission-1
mission-2
mission-3
mission-4
    |
mission-100
```

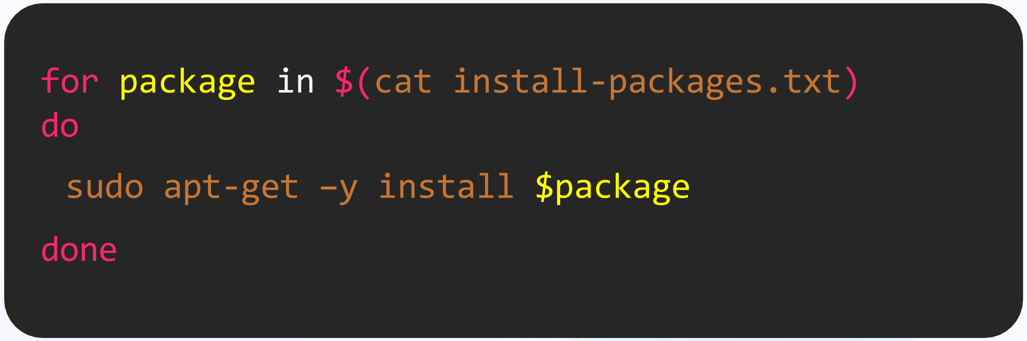
Real life use cases:



```
for file in $(ls)
do

  echo Line count of $file is $(cat $file | wc -l)

done
```



```
for server in $(cat servers.txt)
do

  ssh $server "uptime"

done
```

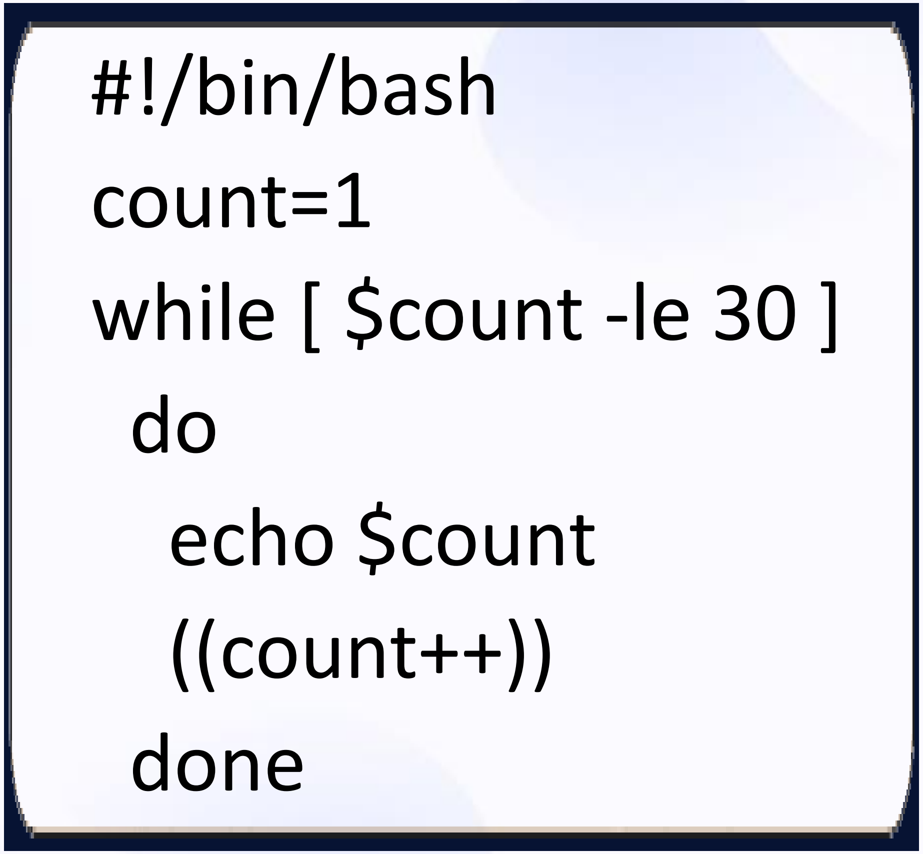


```
for package in $(cat install-packages.txt)
do

  sudo apt-get –y install $package

done
```

while loops

Executes a set of commands so long as the condition(s) is true and stops or do other things otherwise

```
while [ condition ]
do
  command(s)
done
```

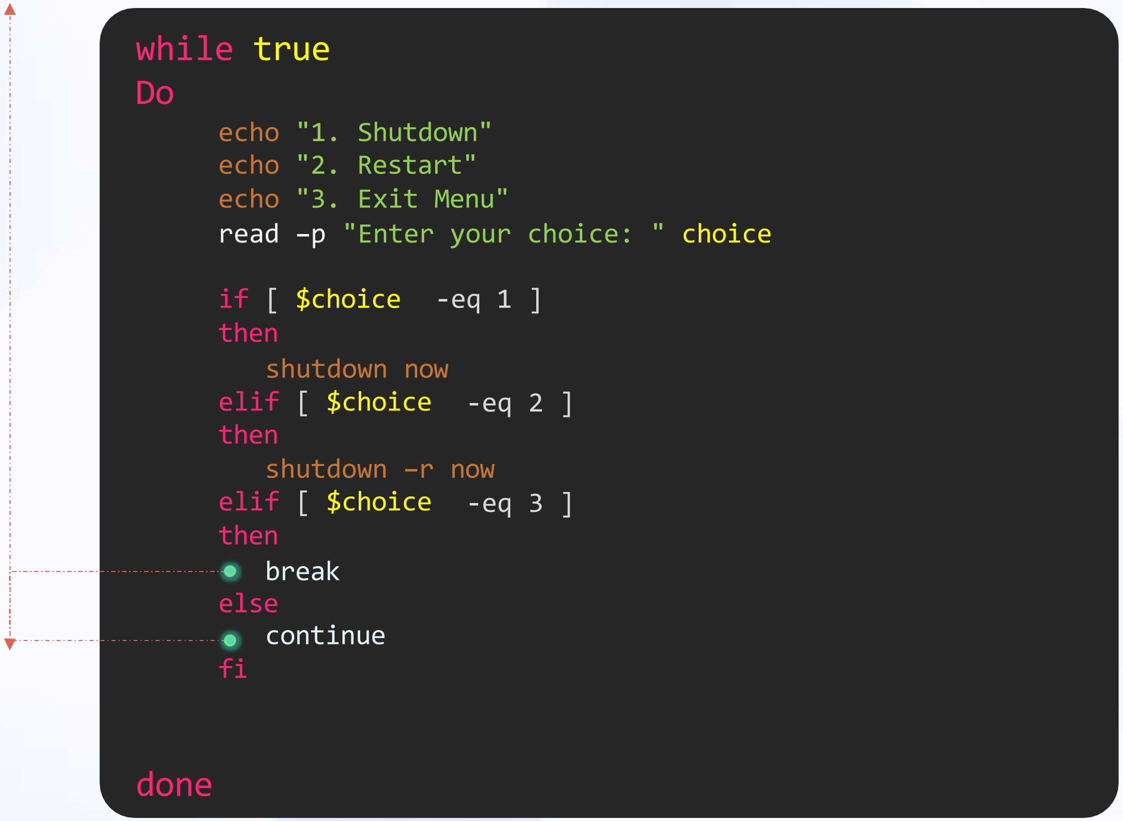


```
#!/bin/bash
count=1
while [ $count -le 30 ]
  do
    echo $count
    ((count++))
  done
```

Infinite loop

```
while [ 1 ]
do
    echo -n "Enter your password“ #no new line
    read input
    if [ $input = "secret" ]
    then
        break  ## break out of the loop
    else
        echo -n "Try again... "
    fi
done
```

Real life use cases:



```
while true
Do
      echo "1. Shutdown"
      echo "2. Restart"
      echo "3. Exit Menu"
      read -p "Enter your choice: " choice

      if [ $choice  -eq 1 ]
      then
         shutdown now
      elif [ $choice  -eq 2 ]
      then
         shutdown -r now
      elif [ $choice  -eq 3 ]
      then
         break
      else
         continue
      fi


done
```

```
while true
do
    echo  "1.  Shutdown"
    echo  "2.  Restart"
    echo  "3.  Exit Menu"
    read -p "Enter your choice: " choice

    if [ $choice  -eq 1 ] then
       echo “shutdown now"
    elif [ $choice             -eq 2   ]
    then
        echo “shutdown -r   now"
    elif [ $choice             -eq 3   ]
    then

        break
     else
        continue
     fi
 done
```

until loop

The until loop execute command(s) as long as a specified condition is false and stops when the condition becomes true, its syntax is as below:

```
until [ condition(s) ]
do
   command(s)
done
```

example

```
#!/bin/bash
counter=0
Until [ $counter –ge 5 ]
do
  echo “The present counter value is $counter”
  ((counter++)
done
echo “The until loop has finished”
```

select loop

When there is the need for a simple menu, the select loop is used, for the user to make choices, its syntax is as below:

```
select choice in choice1 choice2 … choice
do
  echo “your choice is $choice”
done
```

When working with select loop, you use the environment variable PS3 to provide prompts and information to the user

The `select` Command

```
#!/bin/bash

PS3="Select a color: "  # Prompt shown for user selection

select color in Red Green Blue Quit
do
   case $color in
      Red)
         echo "You selected Red."
         ;;
      Green)
         echo "You selected Green."
         ;;
      Blue)
         echo "You selected Blue."
         ;;
      Quit)
         echo "Exiting..."
         break  # Exit the loop if "Quit" is selected
         ;;
      *)
         echo "Invalid option. Please choose a number from 1 to 4."
         ;;
   esac
done
```

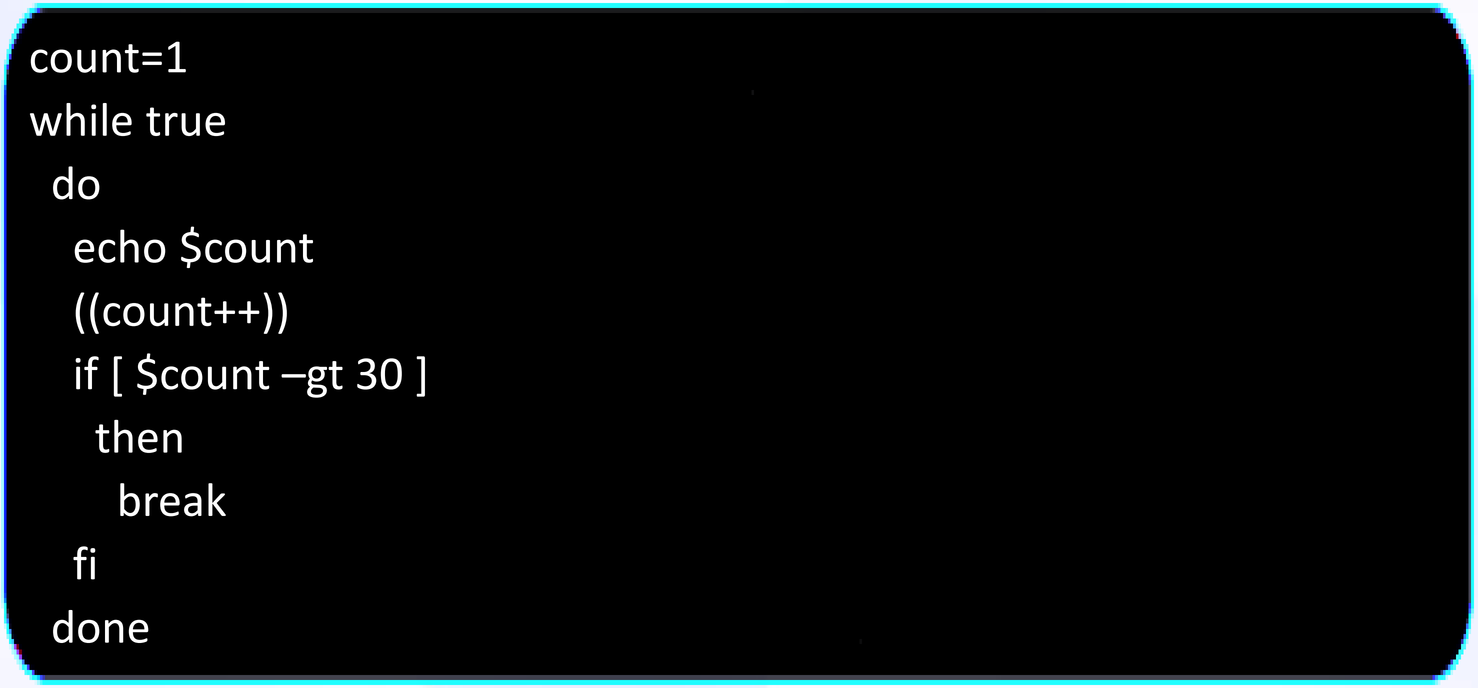
The **select** Command

```
#!/bin/bash
# Scriptname: runit

#  PS3 is the environment variable connected only to the shell
#     select statement
#     Select makes a menu
#      PS3 holds the prompt for the menu option
#      Select header holds a list
#      do / done are like the curly braces of the select
#      The choice made with the number returns the corresponding
#          word to the program
#
echo \$PS3 = start$PS3 End
PS3="Select a progam to execute: "
select program in 'ls -F' pwd date last exit
do
        if [[ $program = exit ]]
        then
           break
        else
           $program
# note that the $program executes a program because the contents of the $ program variable get
translated and run. The contents are a program name
#   so it is nothing special having to do with select.
      fi
done
```

break and continue

- break, continue
  - **break** command terminates the loop
  - **continue** causes a jump to the next iteration of the loop

The break command is used to terminate a loop, it is used with “for” “whle” and “until” loops to exit the loop when certain condition(s) are met:



```
count=1
while true
 do
   echo $count
   ((count++))
   if [ $count –gt 30 ]
     then
       break
   fi
 done
```

The continue statement is used to skip the execution of the current loop and moves to the next command(s). It is also used with loops like the break statement. It also works based on a condition

```
echo “These numbers are not divisible by 2”
for ((i=1; i<=20; i++)); do
   if ((I % 2 == 0)); then
     continue
   fi
   echo  $i
done
```

2. User Interactive

Here, the input is gotten from the user interactively use the read command. With option p you can print an initial message to the screen as below:

```
read -p “message to display on the screen” variable_name
```

Functions

Bash Script

FUNCTIONS

Functions are blocks of code that perform specific tasks. Functions allow you to break down your script/project into modules that are smaller, logical code blocks for simplicity and reusability. its syntax is as below:

```
#define the function
function_name() {
  commands that make up the function
}

#call the function
function_name
```

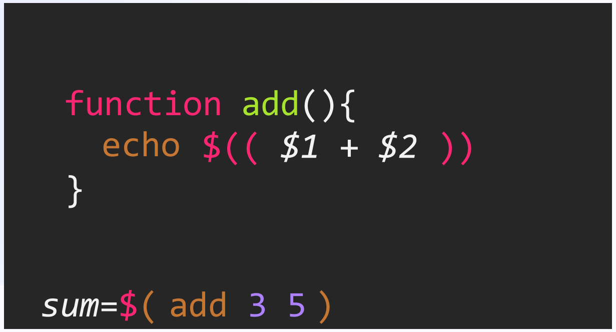
```
greet_nvit() {
  echo “Hello
NNIT”
}


#call greet_nvit
greet_nvit
```

```
sum_game() {
sum=0
while true
  do
    read -p “Enter a score: “ score
    if [[ $score == q ]] #quits if you enter q
    then
      break
    fi
    sum=$(($sum+$score))
    echo “Total Score:  $sum”
  done
}

#call the function
sum_game
```

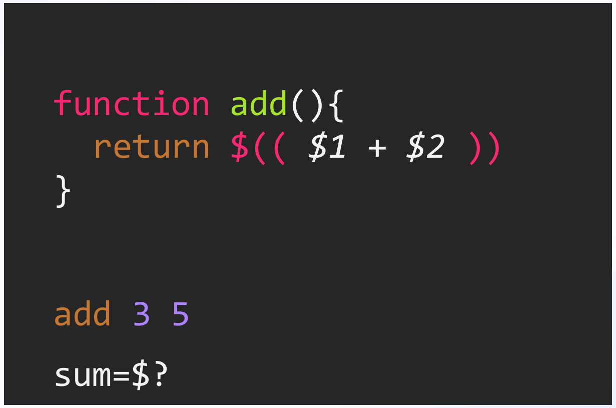
Functions can take command line arguments and can also take inputs from the user:

```
identify() {
  echo “who goes there? State Name”
  echo “Hello $1, what is happening?”
}

#call identify
identify Jethro
```

```
identify2() {
  read -p “who goes there? State Name” name
  echo “Hello $name, what is happening?”
}

#call identify
identify
```



```
function add(){
  echo $(( $1 + $2 ))
}


sum=$( add 3 5 )
```



```
function add(){
  return $(( $1 + $2 ))
}


add 3 5

sum=$?
```

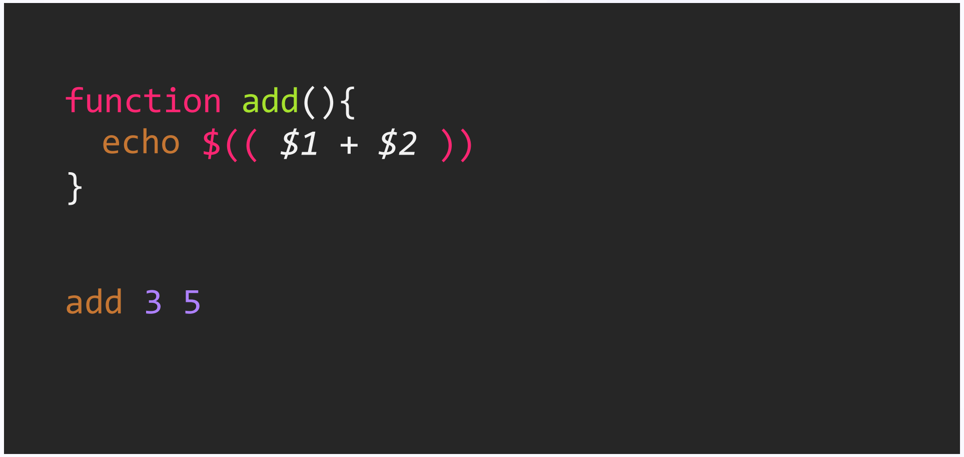
When to use Functions?

- Break up large script that performs many different tasks:
  - Installing packages
  - Adding users
  - Configuring firewalls
  - Perform Mathematical calculations

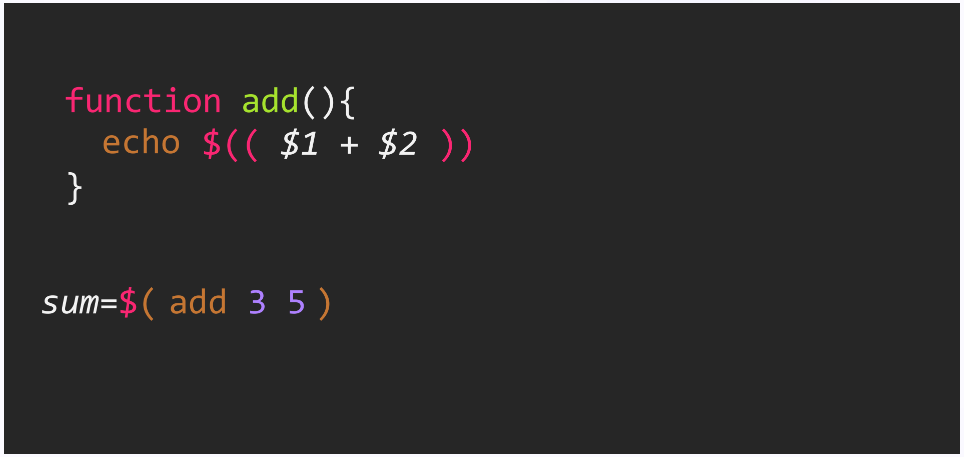
KODEKLOUD

Best Practice

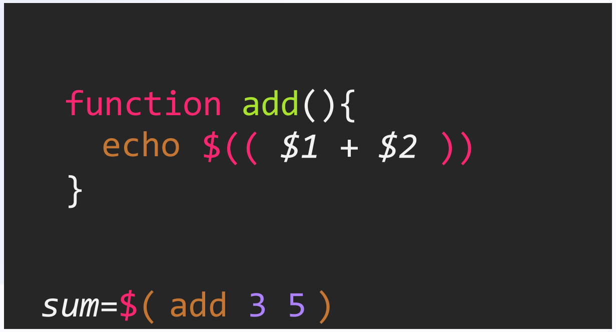
“Always return appropriate exit codes in your script”



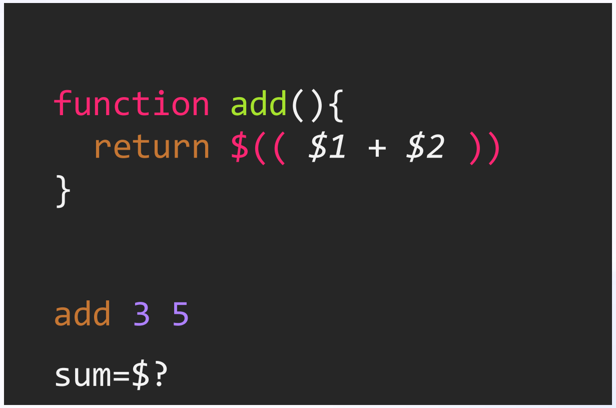
```
function add(){
  echo $(( $1 + $2 ))
}


add 3 5
```



```
 function add(){
   echo $(( $1 + $2 ))
 }


sum=$( add 3 5 )
```



```
function add(){
  echo $(( $1 + $2 ))
}


sum=$( add 3 5 )
```



```
function add(){
  return $(( $1 + $2 ))
}


add 3 5

sum=$?
```

Best Practice

“Always develop scripts in a modular re-usable way using functions”

“Avoid duplicate code”

“Use arguments/parameters to pass in variables”

File IO

Bash Script

File IO

In bash scripting, you can perform file input and output operations using various commands and redirections.

```
wget
http://home.adelphi.edu/~pe16132/csc271/note/scripts/numberit
```

File IO

- read command
  - Reads from stdin unless directed with < or |

```
ls | while read line
do
  echo The line is "$line"
done
```

- Write to a file using redirection >

```
ls | while read line
do
  echo The line is "$line"
done > outputfile
```

- Write to a temp file that is unique – use pid $$

```
 done > tmp$$
```

wget http://home.adelphi.edu/~pe16132/csc271/note/scripts/numberit

grep "pattern" file.txt   # Search for a pattern
sed -i 's/search/replace/' file # Search and replace in a file

```
while IFS= read -r line; do
    echo "Line: $line"
done < input.txt

 echo "Hello, world!" > output.txt  # Write to a file (overwrite)
 echo "More text" >> output.txt     # Append to a file


 cat file1.txt file2.txt  # Concatenate files
 cat file.txt             # Display file content

grep "pattern" file.txt         # Search for a pattern
sed -i 's/search/replace/' file # Search and replace in a file


awk '{print $1}' file.txt  # Print the first column of a file
```

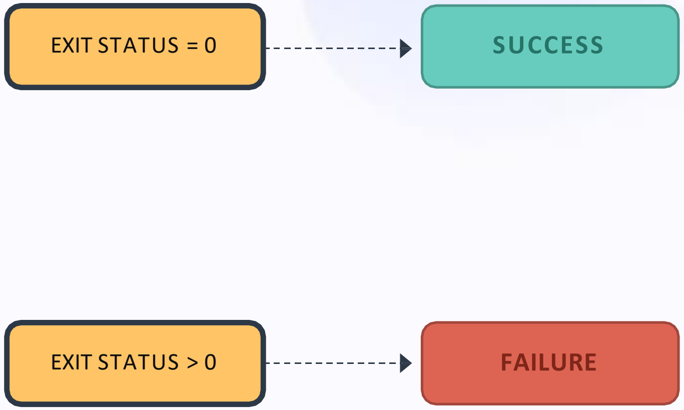
Exit Codes

Bash Script

Exit Codes

```
$ ls
/home /root /tmp
```

```
$ echo $?
0
```



```
$ rocket-status
success
```

```
$ echo $?
0
```

```
$ lss
Failed: command not found
```

```
$ echo $?
127
```

```
$ rocket-status
failed
```

```
$ echo $?
1
```

Exit Status Demo

- All commands return something
- Standard 0 = success and 1 = failure
  - Backwards 0/1 from a true/false boolean

```
grep 'not there' myscript
echo $?
1= failure
grep 'a' myscript
echo $?
0 = success
```

`exit` Command and the `?` Variable

- `exit` is used to terminate the script; it is mainly to used to exit the script if some condition is true.
- `exit` has one parameter – a number ranging from `0` to `255`, indicating if is ended successfully (`0`) or unsuccessfully (nonzero).
- The argument given to the script is stored in the variable `?`

```
wget http://home.adelphi.edu/~pe16132/csc271/note/scripts/ifbigfiles
```

Debugging

Bash Script

Error Handling and Debugging in Bash

Error Handling

Incorporating error handling techniques is critical to ensure the robustness and reliability of bash scripts.

Debugging Tools

Utilizing debugging tools and techniques facilitates the process of identifying and resolving issues in bash scripts.

Debugging

- The Bash shell contains no debugger, nor even any debugging-specific commands or constructs.
- The simplest debugging aid is the output statement, **echo**.
- Set option
  - -n: Don’t run command; check for syntax error only
  - -v: Echo commands before running them
  - -x: Echo commands after command-line processing

Positional Parameters

Positional Parameter	**What It References**
`$0`	References the name of the script
`$#`	Holds the value of the number of positional parameters
`$*`	Lists all of the positional parameters
`$@`	Means the same as $@, except when enclosed in double quotes
`"$*"`	Expands to a single argument (e.g., "**`$1 $2 $3`**")
`"$@"`	Expands to separate arguments (e.g., "**`$1`**" "**`$2`**" "**`$3`**")
`$1 .. ${10}`	References individual positional parameters
`set`	Command to reset the script arguments

```
wget http://home.adelphi.edu/~pe16132/csc271/note/scripts/envvar
```

Calculator using case block

```
case  "$op" in
"+" )       result=$(($x + $y))
               echo $x $op $y = $result;;
"-" )       result=$(($x - $y))
              echo $x $op $y = $result;;
"*" )       result=$(($x * $y))
            echo $x \* $y = $result;;
"/" )       result=$(($x / $y))
            echo $x $op $y = $result;;
* )      echo Unknow operator $op;;
esac
```

Exercise/Example

- Write a function that check whether a user is log on or not (CheckUser.sh)

```
function UserOnline()
{
    if who | grep $1            ## UserOnline takes a parameter
    then
        return 0         ## 0 indicates success
    else
        return 1       ##1 for failure, i.e., offline
   fi
}
if UserOnline $1                   ## function’s return value as condition/test
then
        echo User $1 is online
else
        echo User $1 is offline
fi
```

File Testing

Test Operator	Test True if:
-b filename	Block special file
-c filename	Character special file
-d filename	Directory existence
-e filename	File existence
-f filename	Regular file existence and not a directory
-G filename	True if file exists and is owned nu the effective group id
-g filename	Set-group-ID is set
-k filename	Sticky bit is set
-L filename	File is a symbolic link

File Testing (continued)

Test Operator	Test True if:
-p filename	File is a named pipe
-O filename	File exists and is owned by the effective user ID
-r filename	file is readable
-S filename	file is a socket
-s filename	file is nonzero size
-t fd	True if fd (file descriptor) is opened on a terminal
-u filename	Set-user-id bit is set
-w filename	File is writable
-x filename	File is executable

Trap an Interrupt

- Define the action that will happen when the interrupt occurs using: trap ‘the action to do when the interrupt occurs ‘ the signal:
  - trap 'rm -f /tmp/my_tmp_file_$$' INT
- When the signal arrives, that command will execute, and then it will continue with whatever statement it was processing.
- You can use a function instead of just one command.

  wget http://home.adelphi.edu/~pe16132/csc271/note/scripts/trapper